Progetto Sicurezza dei Dati

Gennaro Pio Rimoli [MAT. 0522501296]

Introduzione

Lo scopo di questo progetto è la realizzazione di una DApp: Applicazione Decentralizzata. Le DApp sono applicazioni che basano il loro funzionamento sulla tecnologia Blockchain e il suo network distribuito, a differenza delle applicazioni tradizionali non esiste alcuna dipendenza con un'entità centralizzata, sono deterministiche e quindi restituiscono lo stesso risultato a prescindere dall'ambiente in cui vengono eseguite e la gestione dei dati avviene in maniera trasparente: tutte le operazioni sono pubbliche e validate attraverso un meccanismo di consenso.

Analisi del progetto

La realizzazione del progetto ha previsto l'uso di uno Smart Contract, ovvero, un codice che gira sulla Blockchain Etherium.

Il progetto inizialmente prevedeva la realizzazione di un'applicazione che permettesse di stipulare assicurazioni di viaggi. Durante l'analisi di questo problema è emerso un problema di natura "filosofica": per poter realizzare questo tipo di applicazione si deve far uso di un'entità terza che "certifica" lo stato del viaggio. Se leggiamo la documentazione di Etherium:

> *"Le dapp possono essere decentralizzate perché sono controllate della logica scritta nel contratto, non da un individuo o da un'azienda."*
> *https://ethereum.org/it/developers/docs/dapps/*

Se avessimo costruito lo Smart Contract e avessimo basato il suo funzionamento su API esterne alla Blockchain veniva meno il concetto di applicazione controllata dalla logica del contratto, infatti, se si ha modo di manipolare il servizio API si ha modo di manipolare il contratto stesso.

Occorre, però, una piccola osservazione, il concetto di determinismo non viene mai violato perché una volta che viene effettuata una chiamata API al servizio, per un determinato viaggio, la risposta viene salvata all'interno della blockchain, quindi chiamate successive al contratto, relative al quel viaggio, produrranno sempre le stesse risposte.

Si è pensato quindi di modificare l'idea iniziale e si è realizzato un gioco: Meta.

Meta è stato realizzato cercando di utilizzare un concetto molto diffuso negli ultimi mesi: gli NFT. Il termine NFT sta per Non-Fungible Token, che in italiano possiamo tradurre come gettone non riproducibile, è un certificato di "proprietà" su opere digitali.

Si è cercato, quindi, di incentrare il progetto sulla creazione di NFT all'interno di un gioco, che chiameremo Item, assieme a questo si è affiancato anche il concetto di Fungible-Token e si è creata una valuta virtuale, Gold.

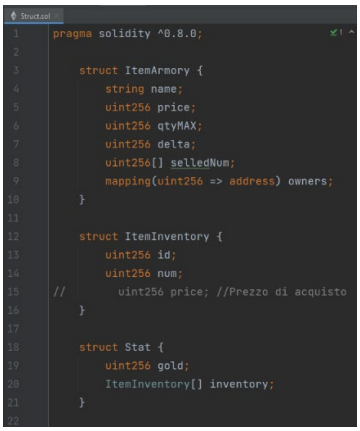
Realizzazione

L'applicativo è stato suddiviso in due parti:

- Backend, scritto in Solidity, si occupa della logica di business e della memorizzazione dei dati.
- Frontend, scritto in Java, si occupa di mostrare all'utente i dati dell'applicazione ne permette l'interazione.

Nella realizzazione del Backend per ottenere un codice di maggiore comprensione e semplice gestione sono state realizzate tre Struct:

- La prima necessaria a tener traccia della quantità di valuta e degli oggetti posseduti da ogni Player
- La seconda contenente le informazioni strettamente necessarie a identificare un oggetto in gioco (ogni oggetto è caratterizzato da: nome, prezzo, quantità massima, quantità venduta, proprietari dell'oggetto)
- La terza ed ultima struttura è necessaria alla realizzazione dell'inventario

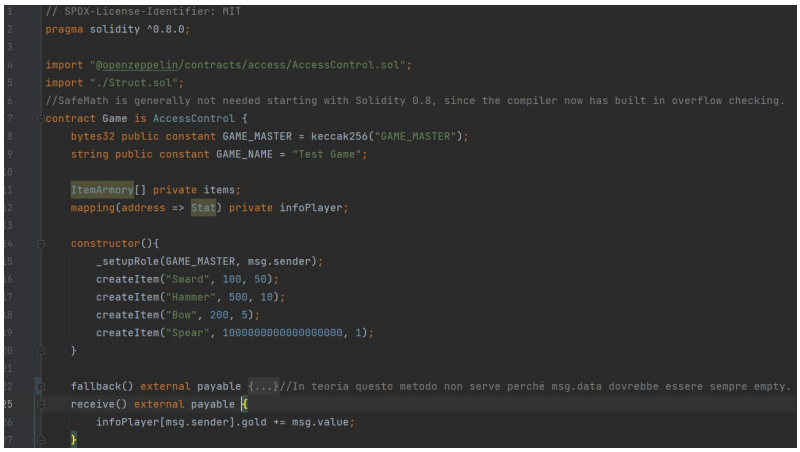


```
pragma solidity ^0.8.0;

    struct ItemArmory {
        string name;
        uint256 price;
        uint256 qtyMAX;
        uint256 delta;
        uint256[] selledNum;
        mapping(uint256 => address) owners;
    }

    struct ItemInventory {
        uint256 id;
        uint256 num;
//        uint256 price; //Prezzo di acquisto
    }

    struct Stat {
        uint256 gold;
        ItemInventory[] inventory;
    }
```

Per evitare problemi di sicurezza noti è stato deciso di utilizzare l'ultima versione disponibile di Solidity, questo ha permesso di evitare l'uso della libreria SafeMath di Openzeppelin.

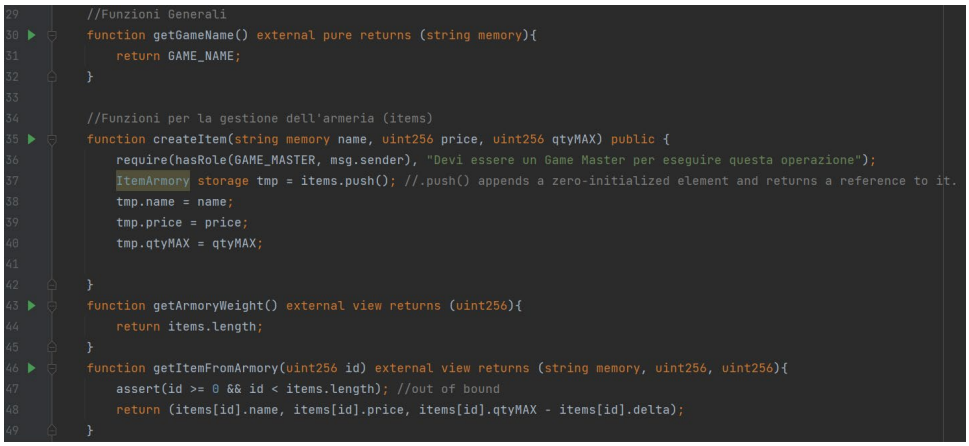
Openzeppelin è una società che mette a disposizione una serie di prodotti per migliorare la sicurezza delle applicazioni sviluppate su Blockchain. Come libreria di questa società è stata utilizzata: AccessControl che permette di gestire una serie di ruoli e evita l'utilizzo di determinate funzioni se non si hanno i giusti permessi.

Anche se Openzeppelin mette a disposizione una serie di utility per essere conformi agli standard sia per la realizzazione di FT (EIP-20), sia per la realizzazione di NFT (EIP-721), sia per la realizzazione di ibridi (EIP-1155), è stato deciso di non implementare tali standard per ridurre al minimo la quantità di Ether necessario ad effettuare il deploy dell'applicativo.



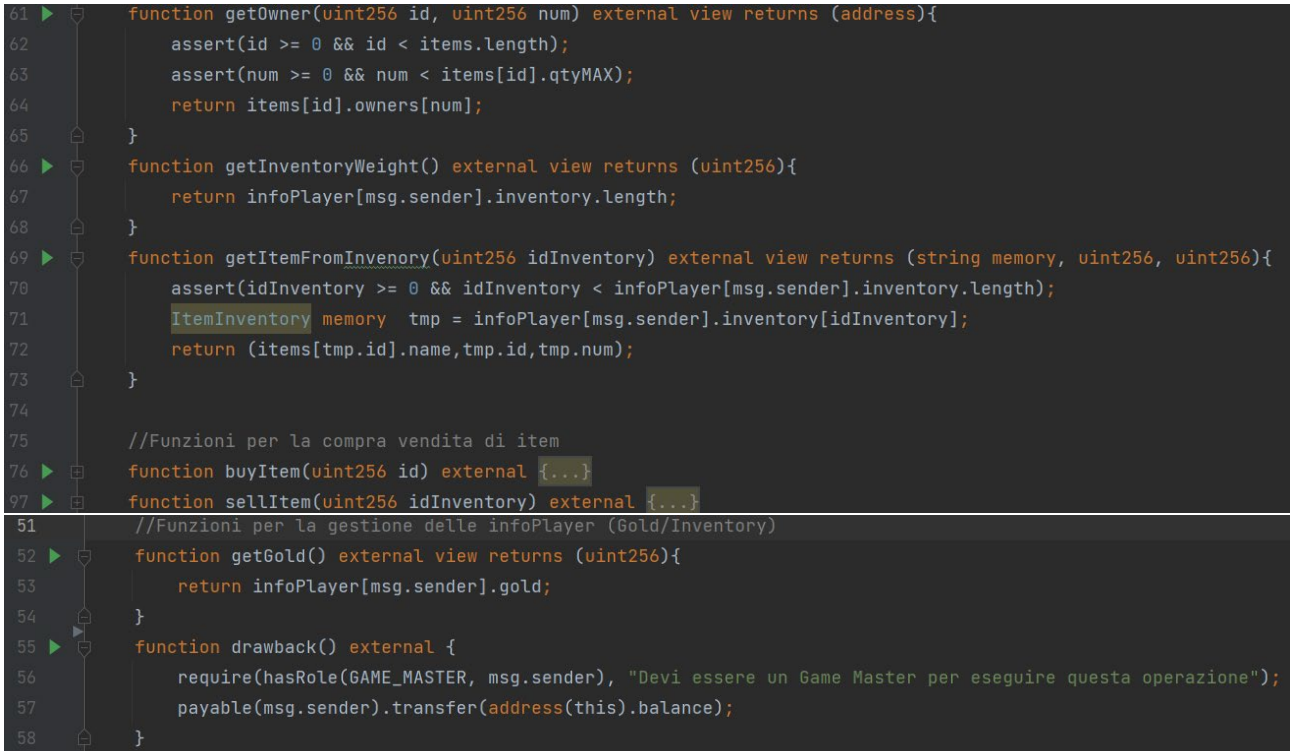
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";
import "./Struct.sol";
//SafeMath is generally not needed starting with Solidity 0.8, since the compiler now has built in overflow checking.
contract Game is AccessControl {
    bytes32 public constant GAME_MASTER = keccak256("GAME_MASTER");
    string public constant GAME_NAME = "Test Game";

    ItemArmory[] private items;
    mapping(address => Stat) private infoPlayer;

    constructor(){
        _setupRole(GAME_MASTER, msg.sender);
        createItem("Sword", 100, 50);
        createItem("Hammer", 500, 10);
        createItem("Bow", 200, 5);
        createItem("Spear", 1000000000000000000, 1);
    }

    fallback() external payable {...}//In teoria questo metodo non serve perché msg.data dovrebbe essere sempre empty.
    receive() external payable {
        infoPlayer[msg.sender].gold += msg.value;
    }
```

Il contratto è stato dotato delle funzioni per accettare pagamenti e poter quindi “comprare” la valuta di gioco trasferendo Ether dal proprio wallet al contratto.



```
    //Funzioni Generali
    function getGameName() external pure returns (string memory){
        return GAME_NAME;
    }

    //Funzioni per la gestione dell'armeria (items)
    function createItem(string memory name, uint256 price, uint256 qtyMAX) public {
        require(hasRole(GAME_MASTER, msg.sender), "Devi essere un Game Master per eseguire questa operazione");
        ItemArmory storage tmp = items.push(); //.push() appends a zero-initialized element and returns a reference to it.
        tmp.name = name;
        tmp.price = price;
        tmp.qtyMAX = qtyMAX;

    }
    function getArmoryWeight() external view returns (uint256){
        return items.length;
    }
    function getItemFromArmory(uint256 id) external view returns (string memory, uint256, uint256){
        assert(id >= 0 && id < items.length); //out of bound
        return (items[id].name, items[id].price, items[id].qtyMAX - items[id].delta);
    }
```

Siccome gli Smart Contract non permettono la restituzione di array e/o strutture (e le letture all’interno della blockchain sono gratuite) l’unico modo per poter permettere al frontend di leggere i dati raggruppati all’interno di insiemi, è dotare il contratto di due funzioni: una che permette di conoscere la grandezza dell’insieme e l’altra che permette di restituire l’elemento in una data posizione. Questa soluzione è stata adottata sia per l’inventario sia per l’armeria.



```
    function getOwner(uint256 id, uint256 num) external view returns (address){
        assert(id >= 0 && id < items.length);
        assert(num >= 0 && num < items[id].qtyMAX);
        return items[id].owners[num];
    }
    function getInventoryWeight() external view returns (uint256){
        return infoPlayer[msg.sender].inventory.length;
    }
    function getItemFromInvenory(uint256 idInventory) external view returns (string memory, uint256, uint256){
        assert(idInventory >= 0 && idInventory < infoPlayer[msg.sender].inventory.length);
        ItemInventory memory  tmp = infoPlayer[msg.sender].inventory[idInventory];
        return (items[tmp.id].name,tmp.id,tmp.num);
    }

    //Funzioni per la compra vendita di item
    function buyItem(uint256 id) external {...}
    function sellItem(uint256 idInventory) external {...}
```

```
    //Funzioni per la gestione delle infoPlayer (Gold/Inventory)
    function getGold() external view returns (uint256){
        return infoPlayer[msg.sender].gold;
    }
    function drawback() external {
        require(hasRole(GAME_MASTER, msg.sender), "Devi essere un Game Master per eseguire questa operazione");
        payable(msg.sender).transfer(address(this).balance);
    }
```

Sono stati infine realizzati i metodi necessari per la compra vendita di oggetti e per permettere all’admin di riscattare l’ether presente nel contratto.

Nella realizzazione del frontend si è utilizzata la libreria Web3J che ha permesso la realizzazione di una classe Wrapper: GameUtil. Questa classe permette, dopo la sua inizializzazione con la chiave privata del wallet con cui si vuole interrogare il gioco, di utilizzare tutti i metodi dello Smart Contract e traduce i return del contratto in variabili gestibili all'interno di Java.

```
public class GameUtil {
    private static final BigInteger GAS_LIMIT = new BigInteger( val: "200000");
    private static final BigInteger GAS_PRICE = Convert.toWei( number: "5", Convert.Unit.GWEI).toBigInteger();
    private static final ContractGasProvider contractGasProvider = new StaticGasProvider(GAS_PRICE, GAS_LIMIT);
    private final Web3j web3j = Web3j.build(new HttpService( url: "HTTP://127.0.0.1:7545"));
    private final Credentials c;
    private final Game gameEngine;
    private final String gameAddr;


    public GameUtil(String gameAddr, String userPKey) {...}

    public void payForGold(double value) throws BlockChainException {...}

    public BigInteger getGold() throws BlockChainException {...}
    public String getOwner(long id, long num) throws BlockChainException {...}
    public List<Item> getArmory() throws BlockChainException {...}
    public List<Item> getInventory() throws BlockChainException {...}

    public void buy(long id) throws BlockChainException {...}
    public void sell(long id) throws BlockChainException {...}


    //Admin Function
    public void createItem(String name, long price, long qtyMAX) throws BlockChainException {...}
    public void updatePlayer(String add, long gold) throws BlockChainException {...}
    public void drawBack() throws BlockChainException {...}
```

Infine si è realizzata un'interfaccia grafica per far interagire l'utente con il contratto.

Conclusioni

Le possibilità di espansione di questo progetto possono essere illimitate, ad esempio, potrebbe essere integrato un sistema di trading (scambio di Items) tra player, aggiungere la possibilità di riconvertire la valuta di gioco, inserire ulteriori informazioni ad ogni item.